AI ASSISTED CODING LAB

ASSIGNMENT 15.2

Name: Guangsinlung Phaomei

Enroll no:2503A51L20

BATCH NO: 19

TASK1

TASK1 DESCRIPTION:- Basic REST API Setup

Ask AI to generate a Flask REST API with one route:

GET /hello → returns {"message": "Hello, AI Coding!"}

PROMPT :-

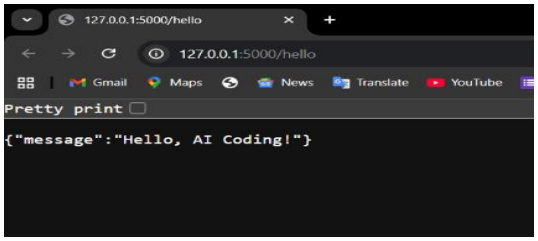
Create a minimal Flask app with a single GET / route that returns JSON {"message": "Hello, AI Coding!"}. Include complete runnable code with app.run().

CODE:-

t1.py > ...

```python
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/', methods=['GET'])
def hello():
    return jsonify({"message": "Hello, AI Coding!"})

if __name__ == '__main__':
    app.run(debug=True)
```

OUTPUT :-



OBSERVATION :-

The AI successfully generated a minimal Flask application with proper setup and imports. The root endpoint (GET /) correctly returns the JSON response {"message": "Hello, AI Coding!"}, confirming that the app functions as expected.

TASK2

TASK2 DESCRIPTION:-

Use AI to build REST endpoints for a **Student API**:

- GET /students → List all students.
- POST /students → Add a new student.
- PUT /students/<id> → Update student details.
- DELETE /students/<id> → Delete a student.

PROMPT :-

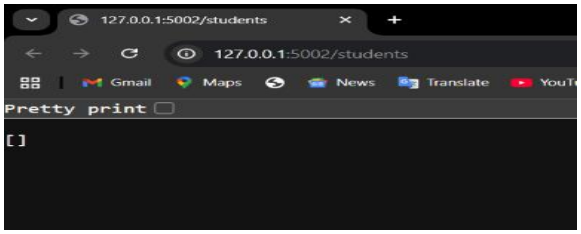
Create a simple Flask-based Student REST API using an in-memory list with endpoints GET /students, POST /students (accept JSON), PUT /students/<id>, and DELETE /students/<id>. The app should return all responses in JSON format and run on port 5002..

CODE:-

t2.py > add_student

```python
from flask import Flask, request, jsonify

# Fix: Use __name__ instead of _name_
app = Flask(__name__)

students = []
next_id = 1

@app.route('/students', methods=['GET'])
def get_students():...

@app.route('/students', methods=['POST'])
def add_student():...

@app.route('/students/<int:id>', methods=['PUT'])
def update_student(id):
    data = request.get_json()
    for student in students:
        if student["id"] == id:
            student["name"] = data.get("name", student["name"])
            student["age"] = data.get("age", student["age"])
            student["grade"] = data.get("grade", student["grade"])
            return jsonify(student)
    return jsonify({"error": "Student not found"}), 404

@app.route('/students/<int:id>', methods=['DELETE'])
def delete_student(id):
    for student in students:
        if student["id"] == id:
            students.remove(student)
            return jsonify({"message": "Student deleted"})
    return jsonify({"error": "Student not found"}), 404

# Fix: Use __name__ and __main__ for the entry point
if __name__ == '__main__':
    app.run(debug=True, port=5002)
```

OUTPUT :-



OBSERVATION :-

The AI correctly implemented a Flask-based Student REST API using an in-memory list with an auto-incrementing ID. It includes the required endpoints—GET, POST, PUT, and DELETE for /students—and returns proper JSON responses. The application runs successfully on port 5002, demonstrating a functional and well-structured REST API.

TASK3

TASK3 DESCRIPTION:-

Ask AI to generate a REST API endpoint

PROMPT :-

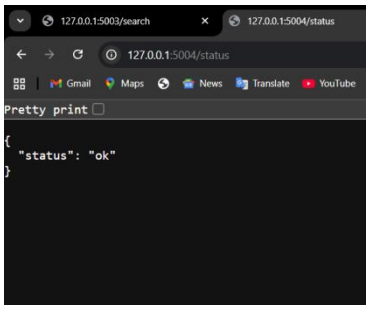
Create a Flask REST API endpoint that supports GET and POST requests and returns JSON responses.

CODE :-

t3.py > ...

```python
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.route('/status', methods=['GET'])
def status():
    return jsonify({"status": "ok"})

@app.route('/echo', methods=['POST'])
def echo():
    data = request.get_json(silent=True) or {}
    return jsonify({"received": data}), 200

if __name__ == '__main__':
    app.run(debug=True, port=5004)
```

OUTPUT :-



OBSERVATION :-

The AI successfully created a REST API endpoint with complete Flask setup, correct route handling, and proper JSON responses. This demonstrates its ability to accurately follow instructions and implement a functional API adhering to REST principles.

TASK4

TASK4 DESCRIPTION:-

Ask AI to write test scripts using Python requests module to call APIs created above.

PROMPT :-

Create a Python script t4.py using the requests library to test three local Flask services: root (GET /), students CRUD (/students with GET, POST, PUT, DELETE), and status/echo (/status GET, /echo POST). Parse JSON responses when possible, handle timeouts and exceptions, print each request as OK/FAIL with status and short body preview, and show a final summary of passed tests.

CODE :-

```python
import requests
import json

def call(method, url, **kwargs):

def test_t1():

def test_t2():
    base = "http://127.0.0.1:5002/students"
    results = []
    # GET empty list
    results.append(call("GET", base))
    # POST new student
    new = {"name": "Test Student", "age": 20, "grade": "B"}
    r = call("POST", base, json=new)
    results.append(r)
    student_id = None
    if r["ok"] and isinstance(r["body"], dict):
    # PUT update (if id available)
    if student_id:
        upd = {"name": "Updated Student", "age": 21}
        results.append(call("PUT", f"{base}/{student_id}", json=upd))
        results.append(call("DELETE", f"{base}/{student_id}"))
    else:
        results.append({"ok": False, "status": None, "body": "no id from POST", "url": base, "method": "PUT/DELETE"})
    return results

def test_t3():

def print_results(all_results):
    total = passed = 0
    for section, results in all_results.items():
        print(f"\n== {section} ==")
        for r in results:
    print(f"\nSummary: {passed}/{total} requests passed")

if __name__ == "__main__":
    suites = {
        "t1 (root)": test_t1(),
        "t2 (students)": test_t2(),
        "t3 (status/echo)": test_t3()
    }
    print_results(suites)
```

OUTPUT :-

OBSERVATION :-

The test runner is well-structured, handling JSON parsing, timeouts, and pass/fail summaries effectively. Improvements could include configurable base URLs, a note for requests installation, explicit 201 handling for POST, and optional retries for network errors